AFRL-RI-RS-TR-2015-074

KEVLAR: TRANSITIONING HELIX FROM RESEARCH TO PRACTICE

UNIVERSITY OF VIRGINIA

APRIL 2015

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

FOR THE DIRECTOR:

/ S /

ROBERT J. VAETH II
Work Unit Manager

/ S /

WARREN H. DEBANY, JR
Technical Advisor, Information
Exploitation and Operations Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* APRIL 2015	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* FEB 2013 – NOV 2014

4. TITLE AND SUBTITLE KEVLAR: TRANSITIONING HELIX FROM RESEARCH TO PRACTICE	**5a. CONTRACT NUMBER** FA8750-13-2-0096
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 62788F
6. AUTHOR(S) Jack W. Davidson, John C. Knight, Michele Co, Jason D. Hiser, Anh Nguyen-Tuong	**5d. PROJECT NUMBER** KEVL
	5e. TASK NUMBER AR
	5f. WORK UNIT NUMBER 01
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Department of Computer Science, University of Virginia, Charlottesville, VA 22904	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RIGA, 525 Brooks Road, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2015-074

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Security weaknesses in DoD information systems remain a major challenge for system stakeholders. We have advanced technology transition for technology developed under the Helix and PEASOUP projects to protect Air Force systems of interests. The result is an asset that, if widely deployed by the DoD, would enable a high level of confidence in the security of DoD systems, in particular, confidence that certain classes of critical vulnerabilities were no longer subject to possible exploitation. Our technology, called Kevlar, includes key security technologies are protective transformations and targeted recovery. The protective transformations are applied to application binaries before they are deployed. Salient features of Kevlar include applying high-entropy randomization techniques, automated program repairs, leveraging highly optimized virtual machine technology, and developing a novel framework for program analysis, transformation and composition.

15. SUBJECT TERMS
KEVLAR, cyber security, binary, lightweight protection, backwards compatibility

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
					ROBERT J. VAETH II
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** (*Include area code*)
U	U	U	SAR	33	N/A

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Kevlar: Transitioning Helix from Research to Practice

1.0 SUMMARY

Security weaknesses in DoD information systems remain a major challenge for system stakeholders. We have advanced the transition of technology developed under the Helix and PEASOUP projects to protect Air Force systems of interests. The result are expected to be an asset that, if widely deployed by the DoD, would enable a high level of confidence in the security of DoD systems, in particular, confidence that certain classes of critical vulnerabilities were no longer subject to possible exploitation.

Weaknesses in software code (such as memory overwriting errors, fixed-width integer computation errors, input validation oversights, and format string vulnerabilities) remain common. Via these weaknesses, attackers are able to hijack an application's intended control flow to violate security policies (exfiltrating secret data, allowing remote access, bypassing authentication, or eliminating services). To mitigate and defend against attacks that seek to exploit such weaknesses, we have developed the Helix architecture. Helix represents the culmination of over 10 years of R&D with support from Defense Advanced Research Projects Agency (DARPA), the National Science Foundation (NSF), the Army and the Air Force, and ongoing support from the Intelligence Advanced Research Projects Agency (IARPA).

We have leveraged the opportunity to take the Helix architecture one step closer to deployment in real systems by developing Kevlar, a completely automatic system for securing applications against attack by well-funded, determined malicious adversaries. Kevlar armors binary programs and protects them from attacks, which could arise from the inevitable vulnerabilities that remain after deployment. The source code is not required nor are any other development artifacts. These features make Kevlar of particular value for software systems that have to be used but for which no development information is available.

The key security technologies used by Kevlar are protective transformations and targeted recovery. The protective transformations are applied to application binaries before they are deployed. Conceptually, these transformations are tailor-made, lightweight "armor" that prevent an attacker from exploiting residual vulnerabilities in a wide variety of classes. Kevlar uses novel, fine-grained, high-entropy diversification transformations to prevent an attacker from successfully exploiting vulnerabilities. To prevent attacks from causing the system to act in undesirable ways, such as crashing or performing unintended actions, Kevlar also provides custom-made, application-specific remediation strategies that may be invoked in the event of an attack.

Kevlar is implemented using dynamic binary transformation. Diversification is applied to the subject binary program prior to deployment. When in use, dynamic binary translation ensures that the functionality of the software as seen by the user is identical to the original program. The mechanism of dynamic binary translation is heavily protected against direct attacks. Kevlar has several major strengths: (a) it is applied to binaries and does not depend on particular languages, compilers, or libraries, (b) it is complementary to other security techniques including inspection, static analysis and testing, (c) it requires no changes to the software development process, and

(d) preliminary performance measurements show that the armoring provided by Kevlar is lightweight incurring modest run-time performance overhead of around 10%.

Salient features of Kevlar include applying high-entropy randomization techniques, automated program repairs, leveraging highly-optimized virtual machine technology, and in general, developing a novel framework for program analysis, transformation and composition.

2.0 INTRODUCTION

Security weaknesses in DoD information systems remain a major challenge for system stakeholders. To mitigate and defend against attacks that seek to exploit such weaknesses, we have developed the Helix architecture. Helix represents the culmination of over 10 years of Research and Development (with support from DARPA, the National Science Foundation, the Army and the Air Force, and ongoing support via IARPA's Stonesoup Program). Salient features of Helix include developing high-entropy randomization techniques, automated program repairs, leveraging highly-optimized virtual machine technology, and in general, developing a novel framework for program analysis, transformation and composition. We propose to transition technology developed under the Helix and PEASOUP projects to protect Air Force systems of interests. We expect the result to be an asset that, if widely deployed by the DoD, would enable a high level of confidence in the security of DoD systems, in particular, confidence that certain classes of critical vulnerabilities were no longer subject to possible exploitation.

The next two sections describe the Helix architecture and our plans to transition this technology so that it can be used to protect current and future Air Force systems. The major component of this effort is to develop Kevlar, a robust easy to use tool for applying the Helix technology to real systems.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Helix

A fundamental problem with current defenses is that they do not redress the asymmetry between attackers and defenders, changing the target system only slowly and reactively in response to attacks. Even approaches that incorporate intrusion detection and tolerance have proven ineffective against determined and well-funded attackers who have at their disposal a growing arsenal of evasive, stealthy, adaptive, polymorphic and metamorphic attacks. To cope with such sophisticated attacks, the Helix architecture uses a combination of defense mechanisms that is both highly effective and metamorphic, i.e., a *high-entropy metamorphic shield*, that presents attackers with a continuously changing attack surface.

Figure 1 provides a high-level conceptual overview of the Helix architecture. An application running in Helix is treated in a holistic way, with information being shared across development, deployment, execution, and response phases in ways that are not possible with traditional architectures.

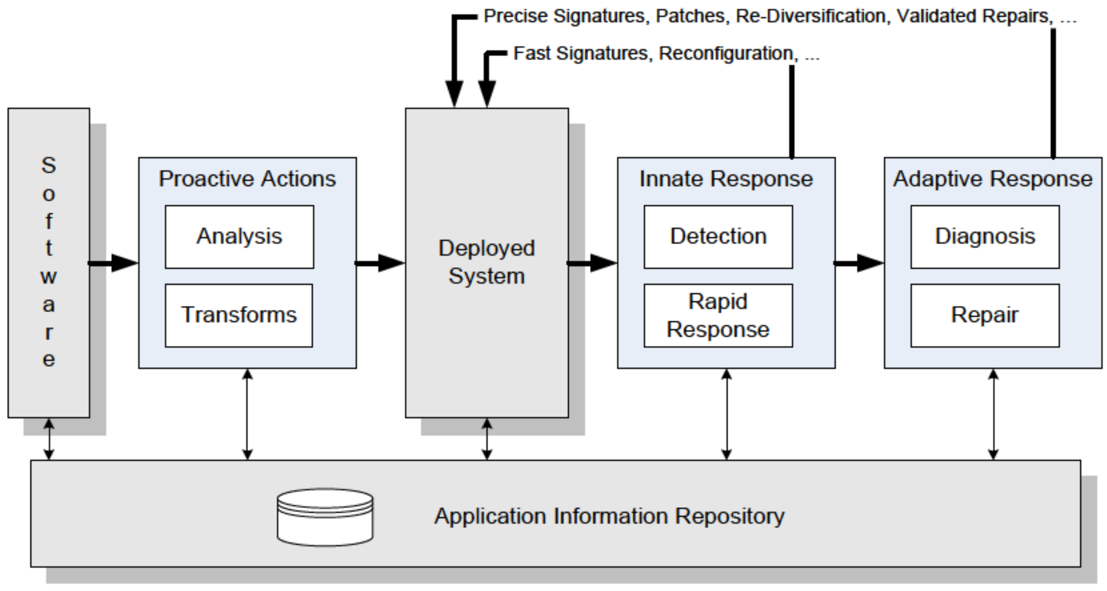
Precise Signatures, Patches, Re-Diversification, Validated Repairs, ...
Fast Signatures, Reconfiguration, ...
Software
Proactive Actions
Analysis
Transforms
Deployed System
Innate Response
Detection
Rapid Response
Adaptive Response
Diagnosis
Repair
Application Information Repository

Figure 1: High-level conceptual overview of the Helix architecture

Instead of viewing the standard tool chain as just a series of steps to transform an application from source code to executable form, we take a more comprehensive view in which program metadata can be deposited in an Application Information Repository (AIR), and subsequently manipulated and enhanced at all phases of a program's lifecycle, to enable the development of novel and accurate security protection algorithms. Starting with applications in source or binary form as input, Helix proactively analyzes and transforms applications to augment them with self-sensing and self-protection capabilities. Helix enables innate and adaptive actions in response or in anticipation to attacks by running applications under control of Strata, a lightweight virtual machine known as a software dynamic translator (SDT). Strata provides the ability to rewrite application code on-demand for deploying security protections and/or dynamically shifting the attack surface of applications.

Helix is a multi-faceted research vision that has lead to many key results. However, as a research project, some ideas are more suited to current, real-world use than others. To transition the best, most deployable of these ideas to practice, from Technical Readiness Level 5 (TRL-5: testing of integrated technology components in representative environment) to Technical Readiness Level 6 (TRL-6: prototype implementation on full-scale realistic systems), we introduce Kevlar. Kevlar directly leverages the following capabilities from the Helix project:

- The concept of analyzing and storing meta information regarding software in the Application Information Repository is key to enabling various security transformations.
- Helix incorporates several novel high-entropy randomization techniques.
- Helix significantly advanced the use of fast dynamic binary rewriting techniques for armoring binaries without requiring the availability of source code.
- Helix leverages the Strata virtual machine technology for transparently augmenting binaries with self-sensing, self-diversification, self-protection and self-repair capabilities.

Overall, Helix provides the intellectual framework for quickly developing and fielding new security transformations. The next section describes how Kevlar's protections exploit Helix-developed capabilities in order to begin an effective transition from research to practice.

3.2 Kevlar Architecture

Kevlar is a completely automatic system for securing applications against attack by well-funded, determined malicious adversaries. It armors binary programs and protects them from attacks which could arise from the inevitable vulnerabilities that remain after deployment. The source code is not required nor are any other development artifacts (such as object files, debugging information, linker maps, etc.) Enabling the rapid development of security transformations and enabling their safe composition are hallmarks of the Kevlar. Kevlar consists of two phases: (1) an offline phase in which Kevlar provides deep analyses on binaries and records results in an intermediate representation database (called the IRDB). Kevlar then uses the database to generate and vet sprockets, i.e. specifications for security transformations; and (2), an online phase in which these sprocket specifications are dynamically applied using Strata, a state-of-the-art dynamic binary rewriter [19, 24].

Figure 2 shows the high-level architecture of the off-line or redeployment portion of Kevlar. Kevlar consists of a static analyzer, called STARS [7], that disassembles x86 binaries, performs extensive static analysis of the binary, and then stores the results of the analysis along with the binary persistent in the IRDB. The IRDB is the repository for all information known or determined about a binary and is the realization of the Application Information Repository described in Section 3.1.

A Kevlar analysis and transformation phase uses information in the IRDB to create new versions of a binary, called variants, where various armoring transformations and remediation policies have been applied. A novel aspect of Kevlar is that, rather than statically rewrite the binary, Kevlar produces programs, called Sprockets, that are used by the software dynamic translation system to transform the original binary into the corresponding variant at run time.

To ensure that the variants produced by Kevlar run appropriately, they are then “vetted” by a tool called BED (Behavior Equivalence Detection) and TSET (Test Suite Evaluation Technology). BED runs each variant using a regression test suite to ensure that the variant produces the same output as the original binary while TSET seeks to measure our confidence levels in the results reported by BED. In addition, BED uses a fault injector to inject faults into the application to determine the effectiveness of the remediation policies generated by Kevlar.

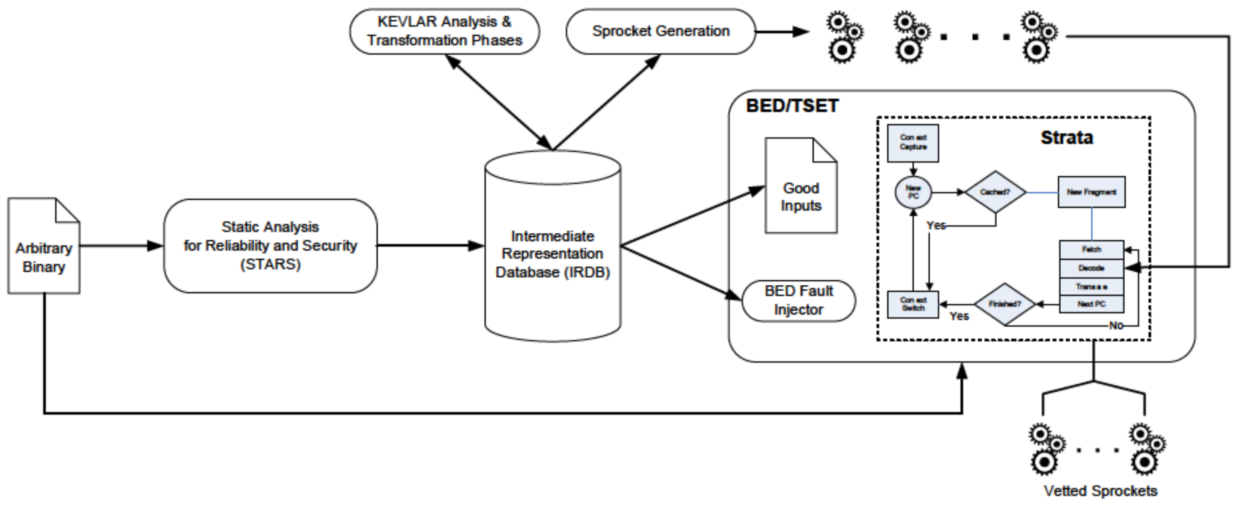

Figure 2: Kevlar Architecture: Offline generation of Sprocket programs

Figure 3 shows a deployed binary that is protected by Kevlar. The vetted Sprocket programs are applied to the binary by the software dynamic translator, Strata. Kevlar has the ability to dynamically select from the set of Sprocket programs to effect temporal change in the protections that are applied. Such changes could be triggered periodically or because an attack has been sensed and remediated and it is desired to add additional protections or to apply different remediation policies.

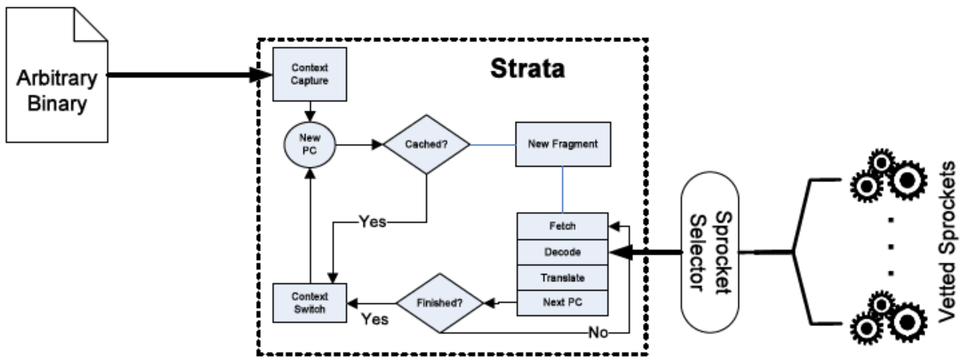

Figure 3: Kevlar Architecture Online Selection of Sprocket program

We highlight several major analysis and security transformations supported by Kevlar. Each transformation can be used in isolation, or composed with other transformations for added protection.

3.2.1 Intermediate Representation Database (IRDB)

To facilitate multiple simultaneous transformations to a program, Kevlar uses an intermediate representation (IR) held in a database, which we term the IR database (IRDB).

The IRDB is similar to the IR for a traditional compiler and the realization of Helix's AIR. It contains information about the program, such as the instructions that make up the program, their addresses, their control and data flow, etc. Furthermore, it contains information about each function including the stack layout, entry points, exit points, etc., the global data layout of the program, targets of indirect branches, etc. Program information is added to the IRDB by various tools including a binary static analyzer called STARS that is discussed in the next section (Section).

Unlike a traditional IR for a compiler, the IR in the database is not guaranteed to be 100% accurate. We realistically assume that information such as perfectly accurate disassembly of the program is not available. We make this assumption to facilitate binary analysis and transformation where such information is rarely available. Issues of imperfect analysis can be compounded if different analyses disagree on information. For example, we use both STARS and the Linux utility `objdump` to populate the list of instructions in the IRDB. The two tools typically agree on instruction start addresses, but occasionally they disagree. The IRDB facilitates the use of conflicting information by supporting conflicting information with a confidence metric.

Another key feature of the IRDB is the ability to "clone" a program. A cloned program is identical to the program it was cloned from, except with a new name, and extra information to track the creation of clones. The primary purpose of a clone is to facilitate programmatic experimentation. For example, suppose that we wish to determine which remediation technique would be effective for a given program. We might choose to clone the program, then instrument the program with the remediation technique for testing.

The clone feature has one other primary purpose: namely we need a "before" and "after" version of the program to support automatic generation of the Sprockets needed to execute the modified program. By tracing a cloned program's hierarchy back to the untransformed program, we can successfully generate Sprockets to represent the changes between the original program and the transformed program. By examining these differences, automatic generation of the Sprockets is fast, efficient and reliable.

Kevlar's IRDB is implemented using PostgreSQL. Measurements show that it is fast and efficient.

3.2.2 Static Analysis for Reliability and Security (STARS)

A key component of Kevlar is STARS (STatic Analyzer for Reliability and Security). It was developed to determine certain security properties of a program in binary form [7]. STARS is implemented as a plug-in to the popular IDA Pro disassembler [6]. The static analyzer currently operates on Linux/x86 binaries, although it can be targeted to any platform that is targeted by IDA Pro. Currently, IDA Pro targets more than 40 processors and operating platforms.

A key function of STARS in Kevlar is the identification of the instructions of the application. As discussed by Debray and Andrews, precisely disassembling a binary is, in general, not a solvable problem [18]. In practice, STARS rarely misidentifies data as code. As noted by Debray and Andrews, such misidentifications would be disastrous for a static code rewriter. Because

Kevlar uses software dynamic translation to transform code, Kevlar is able to tolerate any inaccuracies—we will never rewrite data as code as the rewrite process occurs during the fetch/execute/translate phase of the dynamic translator. That is, only code that should be executed is processed.

The static analyzer analyzes the control flow and data flow of the entire program binary. The analyzer builds a fully pruned SSA (Static Single Assignment) form representation of the program and performs numerous data flow analyses on this representation [3, 23]. The data flow analyses include a simplified type system, in which registers and stack locations are typed as being data pointers, integers, floating-point values, strings, or code pointers.

All information determined by STARS is recorded in the IRDB. Later analysis and transformation phases of Kevlar use this information. During these phases, as additional or more accurate information becomes known, information in the IRDB is updated.

3.2.3 Sprocket Execution Engine

In Kevlar, transformations to the binary are applied dynamically. This approach has several advantages. As mentioned previously, it permits Kevlar to deal with the inaccuracies inherent in the static analysis of binaries. It also permits a single binary to be deployed with transformations applied dynamically to create an ever-changing attack surface—the metamorphic shield. The dynamic rewrites to be applied are specified by sprocket programs rewritten using the Sprocket Program Rewriting Interface (SPRI).

```
Original Program Fragment:
(a) 0x8000  sub esp,20

Rewrite rule:
(1) 0x8000 -> 0xFF00
(2) 0xFFF0 ** sub esp,40
(3) 0xFF01 -> 0x8001
```

Figure 4: Sprocket rewrite rule to change stack frame allocation. For exposition purposes, all instructions are one-byte long.

SPRI and Sprocket Generation SPRI defines simple rewriting rules that come in two forms. The first form, the redirect form, transfers control to a specified target address (lines 1 and 3 in Figure 4). The second form, the instruction definition form, indicates that there is an instruction at a particular location (line 2). The net effect of applying the SPRI rules shown in Figure 4 is to rewrite the instruction `sub esp,20` instruction at address `0x8000` to be `sub esp,40`. The stack layout transformation (described later) uses such rules to transform stack frame allocations.

Together these two types of rules provide the foundation for building a wide range of Sprocket programs. The example shown illustrates the equivalent of a small patch that modifies only 1 instruction. At the other end of the scale, transformations such as ILX (instruction location transformation, described later) seek to rewrite all instructions in a binary.

Despite its conceptual simplicity, manually writing Sprockets in SPRI would be a tedious and error-prone process. Instead, Sprocket developers apply their transformations using a high-level C/C++ API to manage the creation and deletion of program variants, and to manipulate program state, e.g. to insert, delete, or replace instructions and re-route control flow. The API transparently interacts with the IRDB to commit any changes.

With this architecture, the composition of Sprockets is naturally performed by chaining together transformations: one Sprocket encodes its transformation in the IRDB, the next Sprocket then takes as input the new database state, and then effect its own transformations. Kevlar then automatically generates SPRI rules for any program variants by essentially performing a “smart diff” between the IRDB representation of a variant against the IRDB representation of the original binary.

Strata. Once the SPRI rules are generated, Kevlar uses software dynamic translation (SDT) techniques to efficiently execute Sprockets (Figure 4). While we use Strata as our underlying SDT infrastructure, we note that sprockets could be similarly implemented via any SDT tool [2, 12, 14, 19, 20, 22].

Strata dynamically loads an application and mediates application execution by examining and translating an application’s instructions before they execute on the host CPU. Strata operates as a co-routine with the application that it is protecting. Translated application instructions are held in a managed cache called a fragment cache. Strata is first entered by capturing and saving the application context (e.g., program counter (PC), condition codes, registers, etc.) Following context capture, Strata processes the next application instruction. If a translation for this instruction has been previously cached, Strata transfers control to the cached translated instructions.

If there is no cached translation for the next application instruction, Strata allocates storage in the fragment cache for a new fragment of translated instructions. Strata then populates the fragment by fetching, decoding, and translating application instructions one-by-one until an end-of-fragment condition is met (e.g., an indirect branch). As the application executes under Strata’s control, more and more of the applicationâ€™s working set of instructions materialize in the fragment cache.

Implementation of Sprockets requires several simple extensions to a typical software dynamic translator. First, we must modify Strata startup code to read the SPRI rewrite rules (not pictured). Next, Strata’s instruction fetching mechanism is overridden to first check, then read from SPRI rewrite rules as appropriate. Lastly, the next-PC operation is modified to obey any redirection rules that are specified.

Finally, we must take steps to protect Strata itself. To thwart a compromised application from overwriting Strata’s own code or data, we use standard hardware memory protection mechanisms. When executing the untrusted application code, Strata turns off read, write, and execute permission on the pages of memory it uses, leaving only execute (but not write) permission on the code cache. Strata also watches for attempts by the application to change these permissions. Previous work has shown this technique to be effective and cost very little [10].

3.3 Kevlar Protections

The Kevlar toolchain is flexible and powerful. It can dynamically apply a wide range of diversity transformations on a running binary, it can check and enforce various program properties that have been extracted from the binary or specified by an administrator, and it can insert remediation code. The following sections describe some of the Kevlar.

3.3.1 Instruction Location Transformation (ILX)

A powerful diversity technique is to randomize the location of code so an attacker has difficulty precisely locating targets of attack (e.g., entry point to functions, tables of pointers to functions, etc.). For example, most systems now routinely use Address Space Layout Randomization (ASLR) to make exploiting weaknesses difficult [25]. ASLR has several positive attributes. It is cheap to apply incurring little or no run-time overhead, and it can It can be applied to any binary, It is applied automatically—no user intervention or action is necessary.

Unfortunately, ASLR implementations have low entropy. ASLR on a 32-bit architecture only provides 16 bits of entropy. Furthermore, ASLR is not applied universally throughout the address space. Even when using dynamically-linked libraries, it is common for the main program text to start at a known fixed location. Because of these limitations, ASLR-protected code is subject to attack [4, 7, 21].

Instruction Location Transformation (ILX) is a technique that seeks to scatter instructions in a program randomly throughout the address space. In contrast to ASLR, ILX provides 31 bits of entropy on a 32-bit machine. Furthermore, ILX is applied universally to all segments. Thus, the major limitations of ASLR are eliminated.

Figure **5** conceptually illustrates ILX. The top left of the figure shows the control-flow graph of a particular program segment. The compiler and the linker collaborate to produce an executable file where instructions are laid out so they can be loaded into memory when the program is executed. A typical layout of code is shown at the bottom left of the figure. The right side of the figure shows the layout of the code when ILX is applied.

To link instructions together, an ILX sprocket contains a fallthrough map shown at the top right of Figure **5**. This map uses SPRI to specify the execution successor of each instruction in the program.

Together, we call the fallthrough map and randomized instruction locations an ILX sprocket. Figure 6 shows how an ILX program could be created. First, STARS detects the instructions and functions in a program. Next, the program is analyzed for indirect branch targets, call sites, branches, etc. Finally, the reassembly engine uses this information to relocate the entire program with each instruction in a randomized location, and creates the fallthrough map.

Figure 5: ILX code example

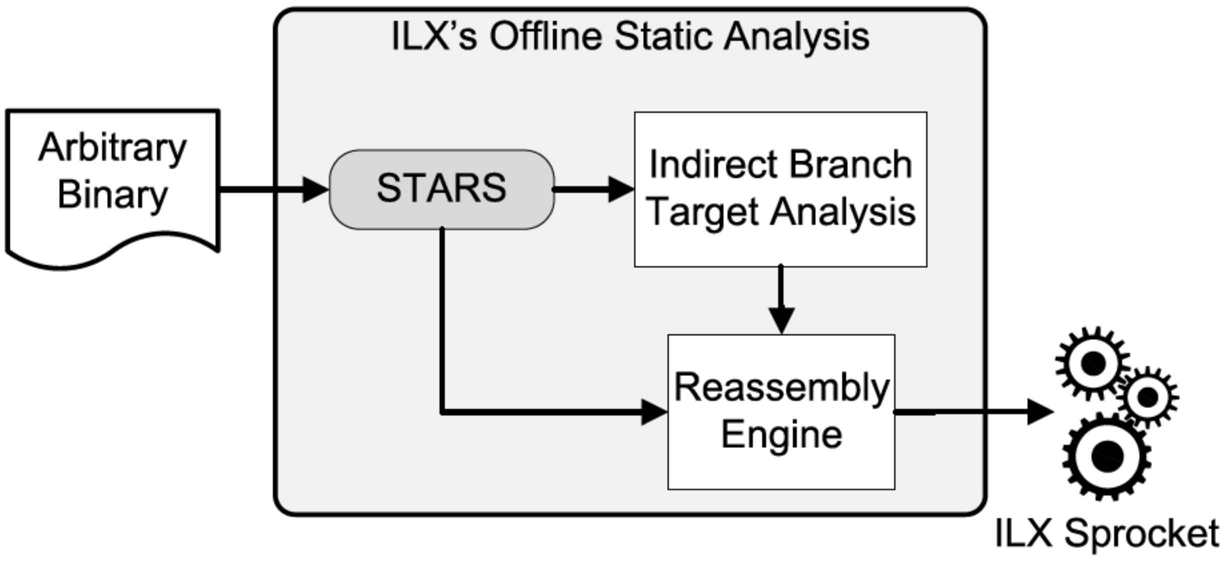
ILX's Offline Static Analysis
Arbitrary Binary
STARS
Indirect Branch Target Analysis
Reassembly Engine
ILX Sprocket

Figure 6: ILX Static Analysis

To execute the randomized program, we use Strata to fetch and execute the instructions from the ILX sprocket. Strata interprets the fallthrough map to fetch and execute instructions on the host hardware. Preliminary results indicate that this process can be made very efficient. The preliminary prototype achieved only 13% runtime overhead on the SPEC2006 benchmark suite. Furthermore, randomly scattering instructions throughout the address space significantly reduces the attack surface for mounting any arc-injection attacks, including attacks based on return-oriented programming techniques. Hiser et al. provides a more complete discussion of the benefits of ILX and full details of its implementation [8].

3.3.2 Stack Layout Transformation (SLX)

A common target of malicious attacks are locations on the stack (e.g., return addresses, frame pointers, function pointers, and critical data). SLX is a transformation that is applied to a running application to dynamically randomize the location of variables on the stack and place canaries to determine if an attack has been attempted.

Transformation of the stack frame layout for a function requires determination of:

1. The current layout of the stack frame, e.g. the addresses and sizes of various stack data objects (incoming arguments, saved registers, return address, local variables, outgoing arguments)
2. The instructions that generate an address of a data objects on the run-time stack.

In principle, if this information were available, the layout of the stack frame could be changed and the instructions that generate stack addresses modified to reflect the new layout. The new layout of the stack frame could be based on any security-relevant criteria, e.g., memory objects could be placed in random order, padding introduced before, after or within the stack, canaries included, variables promoted to the heap, etc. While this information is readily available to the compiler when given a program in source code form, Kevlar must recover this information solely based on the binary representation.

In our approach, static analysis (STARS) is used to determine all the necessary details of the binary program. However, when starting with a binary program, precise determination of the stack layout and the instructions that generate stack addresses for any given function is problematic (indeed, even the very basic notion of a function is problematic at the binary level). Modern compilers employ a wide range of techniques to minimize both the use of storage and program execution time. The result is binary programs with unpredictable structures.

Our approach to determination of the stack layout and the instructions that reference the stack is based on two assumptions about addressing: (a) the predominant mechanism by which instructions access stack variables is through scaled or direct addressing based on an offset indicating the variable starting location, and (b) where indirect addressing is used, that use is for access to variables whose locations can be inferred from previous direct or scaled addressing. Starting with these assumptions, layout inferences are produced using a set of simple heuristics that rely upon additional assumptions concerning: (c) the manner in which the stack is allocated and deallocated, and (d) the general stack frame layout.

The assumptions listed above do not necessarily hold (although assumptions (c) and (d) hold for binaries produced by C/C++ compilers that use the `cdecl` x86 calling convention). Indeed through BED and TSET, the Kevlar toolchain explicitly compensates for any transformations that might rely on erroneous information. Our approach to stack layout transformation is speculative. Initial inferences about the stack are created, and these inferences are then evaluated and refined if necessary to ensure that they preserve the program's semantics.

In our current approach, we limit transformations to placement of memory objects in random order and the introduction of random length padding. Vetting of these transformations is by testing with BED and TSET. Furthermore, we use diversity as an *error amplification* technique to detect bad stack layout inferences. The basic idea behind error amplification is as follows: if a hypothesized stack layout inference is correct, then any semantic-preserving transformations should result in a correct program variant. We can therefore develop a multitude of such transformations, e.g., permutation of the order of variables, and vet each of the variants. If any of them fail, and assuming that our transformation is correctly implemented, we can then not only reject the variant but also the inferred stack layout. Note that this process is the exact opposite of validating transformations by using testing to validate optimizing compiler transformations.

For each detected function in a binary, SLX randomizes the stack layout using an aggressive inference to reorder variables, e.g. using offsets in the disassembly of the program to infer variables. If the tests are passed, we use error amplification before creating the final variant. The layout is randomized a second time and the resulting program tested again. If the tests are passed following the second randomization, a third randomization is effected that reorders the stack elements and places padding between stack objects. If the tests are passed with this randomization, then the transformation is assumed to be satisfactory, and the analysis continues with the next function.

If one or more tests fail during analysis of a function, the inference about the stack layout is abandoned, and a simpler, less aggressive inference is used. The least aggressive inference besides not changing the function at all is one in which the entire stack frame is relocated but the order of variables is left unchanged. Preliminary work suggests that reordering variables is an effective error amplification technique as reordering misidentified variables will most likely result in a program crash. Thus, three rounds of error amplification appears sufficient to vet SLX transformations.

Our current approach has been evaluated on a variety of benchmarks, and the results are promising [16]. The use of BED and TSET resulted in binaries whose functions were transformed with different levels of aggressiveness, ranging from no transforms, to transforms that reordered a subset of the local variables, and in some cases, we were able to infer and reorder all local variables on the stack frame. The ability to reorder stack variables for security purposes is standard in some compilers, e.g., the ProPolice extension to gcc reorders buffers higher in memory than other variables to prevent local overflows [5]. Our results demonstrate that we can enable similar transformations but using only binaries.

3.3.3 Heap Randomization and Transformation (HLX)

Kevlar's Heap Layout Transformation (HLX) provides protection against a variety of common memory errors, such as buffer-overflows, use-after-free, and double-free errors. It achieves these protections by detecting (using STARS analysis) memory allocations within the program and rewriting the allocations to randomly increase the allocation size. HLX also detects memory deallocation sites and maintains a pool of objects that were marked as free. When additional memory is needed (such as the free object pool has become too large), the free pool is checked and objects are randomly selected for deallocation.

Table 1: Example without and with HLX, respectively

`int size = strlen(input);`	`int size = strlen(input);`
`char* newValue =`	`cleanup_free_pool();`
`malloc(size+1);`	`char* newValue =`
`strcpy(newValue,input); \par`	`malloc(random_increase(size+1`
`sprintf(newValue, "%s!\n",`	`));`
`input);`	`strcpy(newValue,input);`
`log("The input is %s",`	`sprintf(newValue, "%s!\n",`
`newValue);`	`input);`
`free(newValue);`	`log("The input is %s",`
	`newValue);`
	`add_to_free_pool(newValue);`

Table 1 provides an example. The left portion shows unprotected source code that allocates a buffer, and uses that buffer to manipulate input. Unfortunately, the code has a off-by-one error, and allocates too few bytes to hold the newly formed string, perhaps because additional characters were were added to the manipulation, but the size of the buffer was not updated. The right side of the figure shows how HLX would transform the program. The amount of memory allocated gets increased by a random amount, and free pool management code is inserted. In this case, the off-by-one error is converted from a possibly crash-inducing bug, into a fully-correct program. While not all programs can be completely repaired, the transform still prevents exploits because an attacker cannot reliably predict where heap items may be located, or what size a buffer might be to predictably overrun the buffer.

Unlike the example, however, HLX provides high-entropy randomization on a binary program where no source code is available, like the rest of Kevlar.

3.3.4 Instruction Set Randomization

A common and very dangerous form of security attack involves exploiting a vulnerability to inject malicious code into an executing application and then cause the injected code to be executed. A theoretically-strong approach to defending against any type of code-injection attack (irrespective of the vulnerability) is to create and use a process-specific instruction set that is created by a randomization algorithm. Code injected by an attacker who does not know the randomization key will be invalid for the randomized processor effectively thwarting the attack.

Kevlar takes advantage of ISR to help defeat these kinds of attacks. ISR uses Kevlar's static analysis and runtime support to identify code locations, and encrypt them during a process' loadeding procedure. Kevlar versions of ISR is based on the first practical version of ISR [9].

Kevlar further extents this technology to monitor the application for dynamically loaded code (shared objects or .dll's) and encrypts that code as it enters the runtime system.

Kevlar's code-injection security can be further enhanced by configuring it as a *metamorphic shield* [13]. The metamorphic shield (MMS) technology not only randomizes the code at program startup, but periodically rerandomizes the code's encryption characteristics as the program is running. Such randomization prevents attackers from exhaustively searching for the encryption key, keeping the program safe from code injection from even the most determined attackers.

3.3.5 PC Confinement (PCC)

ISR provides diversity which prevents malicious code from being injected into the running application, but an attacker may still be able to re-use code that is already in the application to enact security violations, called an arc-injection attack [15 Indirect branches, such as function calls via function pointers and function return instructions, are vulnerable to such an attack if the branch's data is overwritten with a buffer overflow, format string issue, or other program weakness. Table **2** contains an example of a possible arc-injection attack.

Table 2: Example of arc-injection attack

```
void main(){
   auth = authenticate();
   vulnerable_code();
   if(auth) {
      send_secret_data();
   }
}
```

In the table, if the code in `vulenerable_code()` can overwrite the function's return address (or even just part of the function's return address via a partial overwriting attack! [1]), the return instruction can possibly jump anywhere in the program. It may jump to the `system()` function to execute shell commands, or to the send_secret_data() call, to more steathily violate the application's security policy.

Kevlar's ILX feature can defeat many of these attacks by randomizing the application's code. However, Strata's translation and sprocket execution code are at static locations, which may still be targets of arc-injection attacks. Kevlar can protect all statically located code by employing *PC confinement* (PCC). PCC is a type of program shepherding where indirect branches are monitored and only allowed to transfer control to ILX-randomized code [11].

Kevlar's static analysis identifies the location of valid indirect control transfers, and the run-time environment efficiently monitors the execution of indirect branches. Indirect control flow is only allowed if the destination is acceptable. Consequently, PCC and ILX can disallow control transfers to unrandomized code, such as Strata's sprocket execution code, thereby eliminating the vast majority of arc-injection attacks.

3.4 Technology Communication

As part of our approach to transitioning the Helix/Kelvar technology, we participated in a variety of meetings, prepared various publications, and gave several presentations during this period. Our most significant and visible communications are:

- We prepared two posters and handouts for the PACOM exercise in Hawaii. One poster described the Helix/Kevlar technology. A second poster described the demonstration. Handout versions of the posters were also created.
- Anh Nguyen-Tuong made a presentation at the Global Horizons Technical Exchange Meeting held March 25â€“27, 2013 at Rome AFB.
- Dr. Jack W. Davidson travelled to Baltimore to see a GCCS demonstration by Northrup-Grumman at the AFCEA International Cyber Symposium held June 25-27, 2013.
- We published a paper, *A Framework for Creating Binary Rewriting Tools* in European Dependable Computer Conference, 2014.
- We published a paper, *To B or not to B: Blessing OS Commands with Software DNA Shotgun Sequencing* in European Dependable Computer Conference, 2014.
- A student, Sudeep Ghosh, published his dissertation, *Software Protection via Composable Process-level Virtual Machines*
- We published a paper, *What's the PointISA?* in the Second ACM Workshop on Information Hiding and Multimedia Security.
- We made a presentation of the Kevlar technology on September 6, 2013 at the United States Patent and Trademark Office in Alexandria, Virginia at the invitation of the UVa's Vice-President of Research, Thomas Skalak. The presentation was made to a group of venture capitalists and state government officials (including the Governor of Virginia).
- We submitted a provisional patent application called "Datameld".
- Presentation and demonstration of Kevlar at the 2014 International Summer School on Software Protection and Security held in Verona, Italy on July 27â€“August 1, 2014. See `http://issisp2014.di.univr.it/`.

4.0 RESULTS AND DISCUSSION

We present out results (as indicated in the following subsections) for each 3 month period during the project.

4.1 Period 1 – Feb. 2013 to May 2013

4.1.1 Progress Against Planned Objectives

A major objective of this effort is to demonstrate the technical readiness of portions of the Helix technology developed under contract FA8650-10-C-7025 and others. In conjunction with PACOM contractors, we developed a demonstration of Kevlar for an exercise held in Hawaii in early May (May 6–9, 2013). The demonstration was to protect an IRC Chat services component, Anope, that is widely used by the U.S. government. We developed an attack against a vulnerable Anope services module and demonstrated that the protected module was impervious to the attack.

We also provided protected versions of other software including ProFTPD (a FTP server), Apache (web server), xpdf, and bzip2 (a commonly used compression utility), for a Red Team Evaluation. We provided demonstration attacks against xpdf, bzip2, and dumbledore (a synthetic application).

We worked with ReSurgo LLC and MIT Lincoln Labs to develop at test plan for evaluating Kevlar. The result was a test plan document that described the proposed evaluation and the metrics to be used for the exercise.

Sandia provided a report to PACOM and AFRL.

4.1.2 Technical Accomplishments this Period

The major technical accomplishment this period was the demonstration that Kevlar could easily be applied to off-the-shelf binaries by end users (pushbutton protection).

We made improvements to our Strata Program Rewriting Interface (SPRI) to allow dynamic randomization through a random starting address and random selection at load time of the SPRI file to apply.

4.1.3 Improvements to Prototypes This Period

We continue to improve the performance, robustness, and coverage of Kevlar. For example, we fixed some bugs that were exposed during testing. We also extended Kevlar to provide protection to dynamically loaded libraries, to provide dynamic (i.e., moving target) capabilities so each time an application is executed it presents a different attack surface to the adversary.

4.2 Period 2 – May 2013 to Aug. 2013

4.2.1 Progress Against Planned Objectives

A major objective of this effort is to demonstrate the technical readiness of portions of the Helix technology developed under contract FA8650-10-C-7025 and others. Our major goal for this quarter was to take “lessons learned” from the PACOM exercise held in May and incorporate them in the Kevlar roadmap. We have continued to work with MIT Lincoln Laboratories to provide assistance as needed for their evaluation of Kevlar.

4.2.2 Technical Accomplishments this Period

The major technical accomplishments this period were to improve the readiness level of the 32-bit Linux version of Kevlar and to layout the groundwork for a 64-bit Linux version. We also investigated technical issues involved with porting Kevlar to a 32-bit Microsoft Windows platform. Windows XP SP3 was the chosen target platform to evaluate and Visual Studio 2008 was used as the IDE. During this evaluation process it was determined that work effort would be better spent on developing a 64-bit Windows port of Kevlar as the x86-64 Linux port would serve as its foundation.

4.2.3 Improvements to Prototypes This Period

We continue to improve the performance, robustness, and coverage of Kevlar. For example, we fixed some bugs that were exposed while porting the code base to 64 bit. We have successfully implemented a basic stratafier tool to inject the Strata VM into binaries. We have basic implementation of the following Strata capabilities:

- basic translation for x86-64
- signal handling
- system call watching
- threading support
- fork support
- performance optimization

4.3 Period 3 – Aug. 2013 to Nov. 2013

4.3.1 Progress Against Planned Objectives

A major objective of this effort is to demonstrate the technical readiness of portions of the Helix technology developed under contract FA8650-10-C-7025 and others. We have successfully built and configured Kevlar prototypes to armor binaries. We have used regression tests extensively to validate that Kevlar-armored binaries retain the same operational semantics on known good input data. We have also produced documentation and a turnkey â€œpush-buttonâ€ solution to easily armor binaries.

4.3.2 Technical Accomplishments this Period

The major technical accomplishments this period are:

- Testing Kevlar using extensive regression test suites. The availability of test suite servers not only to validate the Kevlar protection but also improves their precision.
- Further supporting the x86-64 architecture
- Work on porting Kevlar to 64-bit Microsoft Windows began. The targeted platform is Windows 7 Enterprise and the IDE chosen is Microsoft Visual Studio 2012. The general porting approach of iterative working models of increasing functionality was designed and work on the straightforward porting tasks such as improving type safety in the Kevlar source code was begun.

4.3.3 Improvements to Prototypes This Period

We have made numerous improvements to the Kevlar toolchain, in particular for x86-64. These include:

- Improved stability of the Strata Virtual Machine that is the core of Kevlar
- Improved stability of the stratafier tool

- Port of the Heap Randomization diversity transformation
- Port of the PC Confinement transformation
- Initiated port of the Instruction Location Randomization technique
- Initiated port of the base IRDB API

In addition we continue to fix bugs we encounter as part of the porting process to x86-64. The net result is to vastly improve the robustness of Kevlar across both the x86-32 and the x86-64 bit versions.

4.4 Period 4 – Nov. 2013 to Feb. 2014

4.4.1 Progress Against Planned Objectives

A major objective of this effort is to demonstrate the technical readiness of portions of the Helix technology developed under contract FA8650-10-C-7025 and others. We have started work investigating the possibility of using Kevlar to protect the Global Command and Control System (GCCS). One major task is understanding the GCCS system so that we can properly protect it. Another major task has been to investigate the feasibility of using Kevlar on a Solaris operating system. Another major task is to port Kevlar to Windows. We have made progress as the Kevlar is now building (i.e., compiling and linking) on Windows. We are now starting the debugging progress.

4.4.2 Technical Accomplishments this Period

The major technical accomplishments this period are:

- Improved Kevlar Testing on x86-64. We have been testing against the Gnome X11 window manager's suite of applications. While bugs still exist, most programs can now be "turn key" protected.
- Further supporting the x86-64 Linux architecture within Kevlar. Our stack randomization protections are now working on this architecture.
- Achieved in-depth estimates of the work required to port Kevlar to x86-32 Solaris.
    - Operating system - We have obtained a Solaris OS, and installed necessary build tools such as gcc, nasm, libelf, postgres, etc.
    - Strata - We have fixed many Solaris/Linux porting issues. Basic dynamic translation support seems to work, however advanced support (signal handling, system call watching, etc.) still needs significant additional work.
    - Stratafier - We have fixed many Solaris/Linux porting issues, and the Stratafication process seems to work on one example. Additional testing is needed.
    - IRDB interfaces - While we have fixed many porting issues, building is still a challenge due to compiler compatibility issues. We anticipate these can be resolved with moderate effort.
    - IDA Pro/STARS - One of the main analysis engines of Kevlar is incompatible with Solaris, and cannot reasonably be ported due to the commercial nature of IDA Pro.

However, we have investigated using a Linux machine to act as a analysis server for Solaris binaries. This seems reasonable, however architecting and implementing the server will require moderate to significant effort.

- Significant work on a x86-32 Solaris port has been achieved, but the first working prototype is still unavailable.
- We are now able to successfully build Strata on a 64-bit Windows platform using Visual Studio. We are now in the process of debugging the code.

4.4.3 Improvements to Prototypes This Period

We have made numerous improvements to the Kevlar toolchain, in particular for x86-64. These include:

- Improved stability of the Strata Virtual Machine that is the core of Kevlar.
- Improved stability and applicability of the Stratafier tool, which now supports Stratfication of executable shared objects.
- Port of the Stack Randomization diversity transformation
- Finished port of the Instruction Location Randomization technique, which includes robustness enhancements for both x86-32 and x86-64.
- Finished port of the base IRDB API

In addition we continue to fix bugs we encounter as part of the porting process to x86-64. The net result is to vastly improve the robustness of Kevlar across both the x86-32 and the x86-64 bit versions.

4.5 Period 5 – Feb. 2014 to May 2014

4.5.1 Progress Against Planned Objectives

A major objective of this effort is to demonstrate the technical readiness of portions of the Helix technology developed under contract FA8650-10-C-7025 and others. We have continued work investigating the possibility of using Kevlar to protect the Global Command and Control System (GCCS). In this period, we made substantial progress as AFRL and Northrup Grumman visited our lab in April to demonstrate the operation of GCCS (in a previous visit by North Grumman, they were not able to demonstrate GCCS because of a configuration problem). We also made substantial progress in modifying Kevlar so it handled x86-64 binaries on Ubuntu (a first goal before moving to x86-64 on Solaris) In addition, substantial progress was made on porting Kevlar to the Windows 8 platform, as we were able to run simple Windows programs under control of the protective virtual machine, Strata.

4.5.2 Technical Accomplishments this Period

The major technical accomplishments this period are:

- Significant refinements and testing on x86-64 Ubuntu platform. All major protections are enabled and work on very large programs. While bugs still exist, most programs can now be "turn key" protected using all Kevlar protections, including ISR and ILR.
- Started work on Kevlar porting to x86-32 Solaris for GCCS system.
    - Operating system - We have been able to install necessary tools (such as gcc, nasm, libelf, postgres, etc.) on a GCCS system.
    - Strata - We have fixed many Solaris/Linux porting issues. Basic dynamic translation support is complete, and advanced support (signal handling, system call watching, etc.) has received significant effort. Most Strata facilities now work correctly on Solaris.
    - Stratafier - We have done additional testing on the Stratafier and found several fundamental issues that prevent the current Stratafier mechanism from being effective and operational. We have redesigned and begun implementation of a Solaris-compatible mechanism.
    - PC-Confinement - A first working version of PC confinement on solaris has been achieved.
    - IRDB interfaces - While we have fixed many porting issues, building was still a challenge due to compiler compatibility issues. To solve this issue, we obtained and installed the SUNSwpro compiler. This has resolved many the compilation issues in the IRDB interface, but some issues continue to remain.
- Significant work on a x86-32 Solaris port has been achieved, and we have a partially working prototype. Before a full release is available, archiving is necessary.
- We are now able to successfully run simple (non-threaded) programs under control of Strata of x86-64 Windows 8 platforms..

4.5.3 Improvements to Prototypes This Period

We have made numerous improvements to the Kevlar toolchain, in particular for x86-64 Ubuntu and x86-32 Solaris. These include:

- Improved stability and performance of the Strata Virtual Machine that is the core of Kevlar.
- Redesign of the Stratafier component for x86-32 Solaris.
- Port of all Kevlar protections to x86-64 Ubuntu.
- Adapting Kevlar for x86-64 Windows 8. Simple (single thread) programs can now run reliably under control of Strata

In addition we continue to fix bugs we encounter as part of the porting process to x86-64. The net result is to vastly improve the robustness of Kevlar across both the x86-32 and the x86-64 bit versions.

4.6 Period 6 – May 2014 to Aug. 2014

4.6.1 Progress Against Planned Objectives

A major objective of this effort is to demonstrate the technical readiness of portions of the Helix technology developed under contract FA8650-10-C-7025 and others. We have continued work investigating the possibility of using Kevlar to protect the Global Command and Control System (GCCS). We are now able to apply some of the Kevlar protections to Solaris executables. The Windows port is coming along. After getting the first version working, we refactored the code so that fit into the overall software architecture in a clean, easy to maintain way.

4.6.2 Technical Accomplishments this Period

The major technical accomplishments this period are:

- Significant refinements and testing on x86-32 Solaris platform. Many major protections are enabled and work on some mid-sized programs, such as bzip2, gedit, etc. While bugs still exist, these programs can now be "turn key" protected using most Kevlar protections, including ISR and ILR.
- We have continued work on Kevlar porting to x86-32 Solaris for GCCS system.
    - Strata - We have fixed most Solaris/Linux porting issues. Basic dynamic translation support and advanced support (signal handling, system call watching, etc.) is complete. All relevant Strata facilities now operate correctly on most Solaris programs.
    - Stratafier - We have finished the redesigned and made progress on an implementation of a Solaris-compatible mechanism. Stratafication of most programs is not possible on Solaris, but further testing is required.
    - ISR - A first working version of ISR, complete with metamorphic shield rekeying, has been achieved on Solaris.
    - IRDB interfaces - The SUNSwpro compiler has resolved many of the compilation issues in the IRDB interface. However, some issues remained. We have resolved many porting issues, and the IRDB interfaces are working for many programs. Further testing and debugging remain necessary.
- Significant work on a x86-32 Solaris port has been achieved, and we have a working prototype that includes many Kevlar defenses.
- We are now working to understand threading on Windows so the relevant Strata code can be adapted appropriately.
- We have further protected a binary from the GCCS system, further testing on GCCS remains.

4.6.3 Improvements to Prototypes This Period

We have made numerous improvements to the Kevlar toolchain, in particular for x86-32 Solaris. These include:

- Redesign and Reimplementation of the Stratafier component for x86-32 Solaris.
- Port of many Kevlar protections, including ISR and PC-Confinement to x86-32 Solaris.
- First functional prototype of Kevlar for x86-32 Solaris.

In addition we continue to fix bugs we encounter as part of the porting process to x86-64. The net result is to vastly improve the robustness of Kevlar across both the x86-32 and the x86-64 bit versions.

4.7 Period 7 – Aug. 2014 to Nov. 2014

4.7.1 Progress Against Planned Objectives

A major objective of this effort is to demonstrate the technical readiness of portions of the Helix technology developed under contract FA8650-10-C-7025 and others. We have continued work investigating the possibility of using Kevlar to protect the Global Command and Control System (GCCS). We are now able to apply many of the Kevlar protections to Solaris executables. We have protected two GCCS executables from the track management system (TMS). We the protected executables to AFRL for evaluation. We have also protected many other executables in GCCS, and tested them internally. All seem to function properly. We have also advanced the Windows port of the basic Kevlar.

4.7.2 Technical Accomplishments this Period

The major technical accomplishments this period are:

- Significant refinements and testing on x86-32 Solaris platform. Most major protections are enabled and work on some mid-sized and large-sized programs, such as bzip2, gedit, gnome-terminal, etc. Most programs can now be "turn key" protected using most Kevlar protections, including ISR and ILR.
- We have continued work on Kevlar porting to x86-32 Solaris for GCCS system, and have succeeded in protecting many of the TMS programs.
- We are achieved success on handling threaded applications on the Windows OS.
- We are now working to understand exception handling on Windows so the relevant Strata code can be adapted appropriately.

4.7.3 Improvements to Prototypes This Period

We have made numerous improvements to the Kevlar toolchain, in particular for x86-32 Solaris. These include:

- Porting of several IRDB faciltiies, including *fast spri* and *preLoadedILR* to improve loading times.
- Port of many Kevlar protections, including final porting of the ILR transformation.
- First fully functional prototype of Kevlar for x86-32 Solaris.

In addition we continue to fix bugs we encounter as part of the porting process to x86-64. The net result is to vastly improve the robustness of Kevlar across both the x86-32 and the x86-64 bit versions.

4.8 Results Discussion

During the many periods of this project we have done much to understand and promote the possible transition of Helix/Kevlar. First, various meetings, publications and presentations have helped us connect with possible transition partners and taught us the constraints customers may have for a transitionable technology. Our porting to Windows and Solaris platforms has helped us both gain an deeper understanding these issues and constraints, as well as made our prototype technology more attractive to potential partners. We have learned much, namely that different operating systems have different default compilers, which can provide significant challenges to a tool that works on the compiler's output (a binary program). In particular, some of our Linux-based tools assumed a particular calling convention, and different systems use different calling conventions. Abstracting the calling convention as much as possible eases technology transition.

5.0 CONCLUSIONS

Security weaknesses in DoD information systems remain a major challenge for system stakeholders. We have advanced the transition of technology developed under the Helix and PEASOUP projects to protect Air Force systems of interests. The result are expected to be an asset that, if widely deployed by the DoD, would enable a high level of confidence in the security of DoD systems, in particular, confidence that certain classes of critical vulnerabilities were no longer subject to possible exploitation.

We have leveraged the opportunity to take the Helix architecture one step closer to deployment in real systems by developing Kevlar, a completely automatic system for securing applications against attack by well-funded, determined malicious adversaries. Kevlar armors binary programs and protects them from attacks which could arise from the inevitable vulnerabilities that remain after deployment. The source code is not required nor are any other development artifacts. These features make Kevlar of particular value for software systems that have to be used but for which no development information is available.

During this project we have done much to understand and promote the possible transition of Helix/Kevlar. First, various meetings, publications and presentations have helped us connect with possible transition partners and taught us the constraints customers may have for a transitionable technology. Our porting to Windows and Solaris platforms has helped us both gain an deeper understanding these issues and constraints, as well as made our prototype technology more attractive to potential partners. We have learned much, namely that different operating systems have different default compilers, which can provide significant challenges to a tool that works on the compiler's output (a binary program). In particular, some of our Linux-based tools assumed a particular calling convention, and different systems use different calling conventions. Abstracting the calling convention as much as possible eases technology transition.

6.0 REFERENCES

1. S. Alexander. Defeating compiler-level buffer overflow protection. *J-LOGIN*, 30(3):59–71, 2005.
2. Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.
3. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, October 1991.
4. Tyler Durden. Bypassing PaX ASLR protection. *Phrack Magazine*, 0x0b(0x3b), 2002.
5. H. Etoh. GCC extension for protecting applications from stack-smashing attacks (ProPolice), 2003.
6. Hex-Rays. IDA Pro. `http://www.hex-rays.com/products/ida/index.shtml`.
7. Jason Hiser, Clark L. Coleman, Michele Co, and Jack W. Davidson. MEDS: The memory error detection system. In Fabio Massacci, Samuel T. Redwine Jr., and Nicola Zannone, editors, *Proceedings of the First International Symposium on Engineering Secure Software and Systems ESSoS*, volume 5429 of *Lecture Notes in Computer Science*, pages 164–179. Springer, 2009.
8. Jason D. Hiser, Anh Nguyen-Tuong, Michele Co, Jack W. Davidson, and Matthew Hall. ILR: Where'd my gadgets go? *IEEE Symposium on Security & Privacy*, pages 571–585, May 2012.
9. Wei Hu, Jason Hiser, Daniel Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the Second International Conference on Virtual Execution Environments*, pages 2–12, Ottawa, Canada, June 2006. ACM Press.
10. Wei Hu, Jason D. Hiser, Daniel Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 2–12. ACM Press New York, NY, USA, 2006.
11. Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *Eleventh USENIX Security Symposium*, august 2002.
12. Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

13. A. Nguyen-Tuong, A. Wang, J.D. Hiser, J.C. Knight, and J.W. Davidson. On the effectiveness of the metamorphic shield. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pages 170–174. ACM, 2010.

14. Mathias Payer and Thomas R. Gross. Generating low-overhead dynamic binary translators. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, pages 22:1–22:14, New York, NY, USA, 2010. ACM.

15. J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *Security & Privacy, IEEE*, 2(4):20–27, 2004.

16. Benjamin Rodes, Anh Nguyen-Tuong, Jason D. Hiser, John C. Knight, Jack W. Davidson, and Michele Co. Against stack-based attacks using speculative stack layout transformation. In *Proceedings of the Third International Conference on Runtime Verification*, RV'12, pages 308–313, Berlin, Heidelberg, September 2012. Springer-Verlag.

17. G.F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib (c). In *2009 Annual Computer Security Applications Conference*, pages 60–69. IEEE, 2009.

18. B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering*, pages 45–54, 2002.

19. K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society.

20. Kevin Scott and Jack Davidson. Strata: A software dynamic translation infrastructure. In *IEEE Workshop on Binary Translation*, September 2001.

21. Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

22. Swaroop Sridhar, Jonathan S. Shapiro, and Prashanth P. Bungale. HDTrans: a low-overhead dynamic translator. *SIGARCH Computer Architecture News*, 35:135–140, March 2007.

23. Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

24. Daniel. Williams, Wei Hu, Jack W. Davidson, Jason D. Hiser, John C. Knight, and Anh Nguyen-Tuong. Security through diversity: Leveraging virtual machine technology. *IEEE Security & Privacy*, 7(1):26–33, Jan.-Feb. 2009.

25. Jun Xu, Z. Kalbarczyk, and R.K. Iyer. Transparent runtime randomization for security. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems*, pages 260–269, oct. 2003.

6.1 List of Acronyms

ASLR	- Address Space Layout Randomization
AIR	- Application Information Repository
BED	- Behavior Equivalence Detection
DARPA	- Defense Advanced Research Projects Agency
DoD	- Department of Defense
GCCS	- Global Command and Control System
HLT/HLX	- Heap Layout Transformation
IARPA	- Intelligence Advanced Research Projects Agency
ILT/ILX	- Instruction Location Transformation
IR	- Intermediate Representation
IRDB	- Intermediate Representation Database
NSF	- National Science Foundation
SDT	- Software Dynamic Translator
SPRI	- Sprocket Program Rewriting Interface
STARS	- Static Analyzer for Reliability and Security
SSA	- Static Single Assignment
TMS	- Track Management System
TRL	- Technical Readiness Level
TSET	- Test Suite Evaluation Technology